Search in heavily annotated language corpora using an XML database

Georgios Komninos

Master of Informatics

School of Informatics
University of Edinburgh
2014

Abstract

The NITE XML Toolkit (NXT) is an open source system appropriate for manipulating language corpora in XML form. It is capable of dealing efficiently with multimodal and cross-annotated data sets. In order to query over natural language data NXT comes together with its own well-designed logical query language, NITE Query Language (NQL). However, its implementation is old and hence slow, memory intensive and impossible to maintain. Moreover, it lacks some key operators such as sisterhood, which would be very useful for linguistic engineers. Since the latest implementation of NQL XML databases have been invented. The purpose for this M.Sc. project is on the one hand developing a modern implementation of NQL on top of an existing XML database and on the other hand completing the language by implementing a set of missing operators.

Acknowledgements

I would like to express my appreciation to my supervisor, Professor Henry Thompson. Without his guidance, suggestions and feedback it would be impossible for me to complete the project.

Moreover, special thanks go to Jonathan Kilgour for helping me get all the resources required for the project and solve some issues that came up during the process.

Finally, I would like to express my gratitude to my parents for their love and support, which throughout the years has guided me to become what I am today.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Georgios Komninos*)

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Linguistic engineers and researchers commonly have to use data sets that consist of annotations for different phenomena that are linked with the base text or any other signal in various ways. Moreover, different tools use different formats to express their data and consequently data cannot be shared across different platforms. In response to that, the NITE XML Toolkit (NXT) is a set of libraries and tools that allow native representation, manipulation, query and analysis of such data.

NXT comes together with its own integrated logical query language, the NITE Query Language (NQL). NQL's strong points include that is allows researchers to perform queries that are significant in linguistic engineering, such as time overlap and range queries, easily and has a clear and flexible design. However, since it was initially designed so that the processed queries involved only one observation at a time (an observation could be a signal, dialogue or text) its performance drops significantly when more observations are involved. The attractive features of NXT and NQL have urged us to look for alternative ways of implementing the language, so that it can handle large scale analysis in an efficient way.

1.1 Contribution

In this project we will reimplement the NITE Query Language, attempting to target the current implementation's problems. We intend to take advantage of tools that on the one hand guarantee high performance and on the other hand have become standards in their domain, meaning that there is common familiarity with them.

Since the last implementation of NQL, XML databases have been invented. An XML database is a data persistence system that stores XML formatted data, allowing

to query, serialize and export the data. The core idea of our new implementation is to develop it on top of an existing XML database, in order to benefit from the capability of such a system to evaluate queries efficiently and the fact that such systems support XML query-related standards such as XPath and XQuery.

1.2 Outline

The rest of this dissertation is organized as follows. In Chapter 2 we present the background and related work to our project. We make a brief presentation of NXT, focusing on the data model it uses and the current implementation of its query language. In Chapter 3 we will describe the procedure we followed in order to select which XML database system to use. Chapter 4 will contain a detailed description of our implementation, its strengths and its limitations. In Chapter 5 we will present the experiments we made to test our work and compare it to the old implementation. Finally, in Chapter 6 we discuss the conclusions and the potential future work related to our project.

Chapter 2

Overview of the NITE XML Toolkit

In this chapter we will present NXT, highlighting its salient features which are the main motivation behind this project. Our presentation will focus on the data model of the system, its query language and how these two important aspects work together. Finally, we will present the most important alternative tools for annotating natural language data, in comparison with NXT.

2.1 NITE XML Toolkit

The motivation behind the creation of NXT was that most existing tools could either handle timestamped data, allowing their timing and sequencing, or work with data with some specific linguistic structure. The NITE XML Toolkit was initially introduced in [5] as a set of libraries that could be used by developers when creating tools that deal with natural language data that is both timed and has linguistic structure.

Apart from the libraries the toolkit contains a set of useful components. Its data handling model, categorizes data objects in an intuitive and self-explanatory way. In order to enable data analysis the toolkit provides its own query language, designed accordingly so that it may serve any need that a user may have. Moreover we have a set of tools that can be used to create GUIs on top of the libraries along with some such sample applications and utilities.

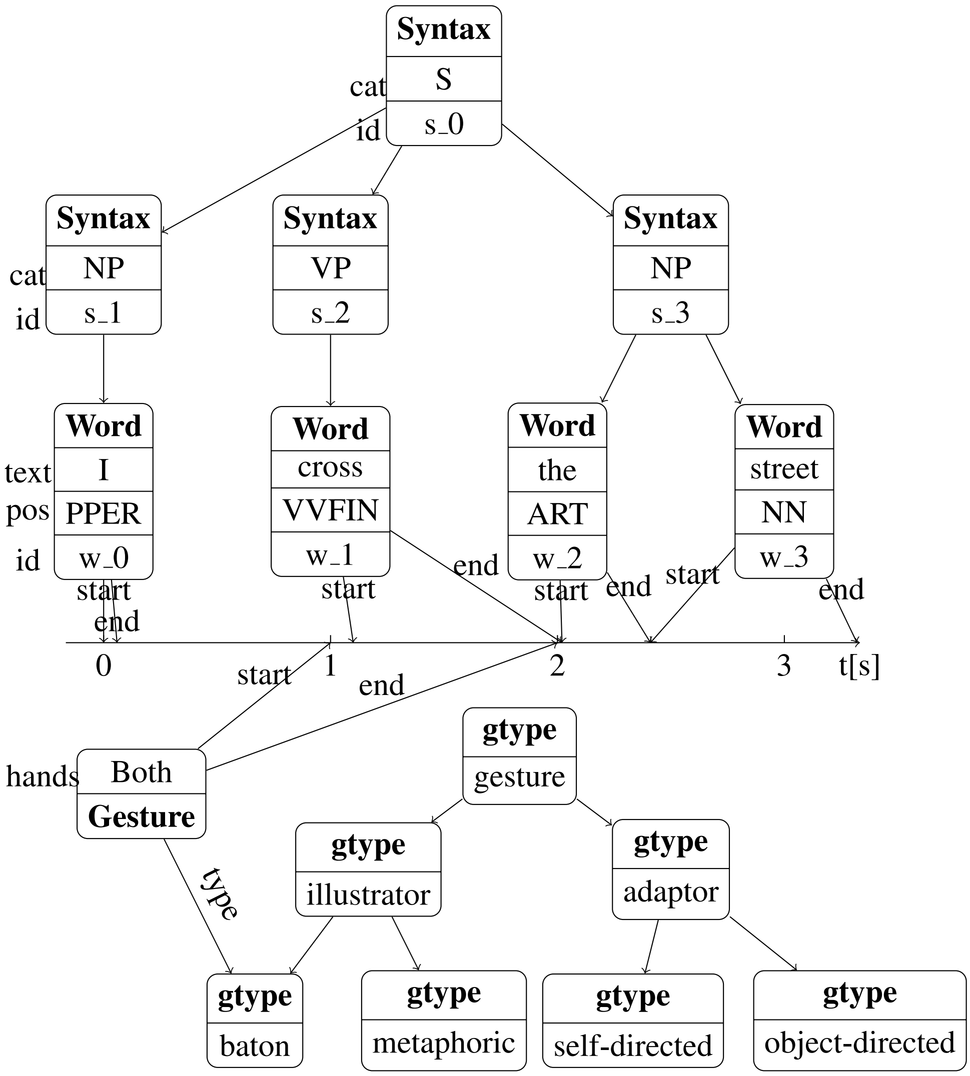
2.1.1 Data model

The toolkit's data model, one of the most important aspects of the platform was presented in more detail in [4]. According to it, the *corpus*, which practically refers to the

data set, is constructed by a set of *observations*. An *observation* stores the data that fully describe one interaction, which may be a dialogue or a group discussion. Finally, observations are recorded by one or more *signals*, which typically is audio or video data recorded during the interaction but can also contain data from various sources such as blood pressure monitors, lie detectors and whatever each research requires.

In order to handle data NXT uses the notion of data objects, which refer to single data elements. A data object has a type, which is a simple string along with four kinds of properties: 1)*attribute-value pairs* (the attribute is a string and the value is either a string or a number) provide further information for the object. 2)*Timing information* refers to start and end times of the signal (all objects of an observation are time-aligned). 3)*Relationships* refer to links between objects. The most important type of relationship is a parent-child relationship, that defines a structural relation between the two objects. It is important to clarify that NXT parent-child relationships are not the same thing as parent and child nodes that exist in typical XML documents and we will see an example to explain the difference. Other relationships can be any link between two objects and may hold its own semantics. Finally, the 4)*textual content*, which may be be present only if an object is not related to other object as their parents.

We should make some more observations about the data model. Firstly, paths generated by objects and parent/child relationships cannot contain cycles, while such a constraint does not exist for other relationship pointers. Secondly, most commonly a data object refers to a single annotation, such as a word or gesture. There is a chance however that they can represent a universal element related to the observation, such as things that the speakers may point to during a dialogue and such objects do not have timings. Finally, it is very important that objects can be ordered in two different ways, one with reference to temporal ordering (in which an object inherits its timings from its descendant) and one based on child list ordering.



In order to make the data model clear, the above figure shows a simple example of a coded piece of data. We firstly have a syntax object that contains a whole sentence and therefore is of category S, as defined by its *cat* attribute. The sentence is composed by smaller syntactic categories such as noun phrases (NP) and verb phrases (VP) and because of that there is a parent/child link between them. Similarly, simple syntactic types are constructed by single words. Words have their timing attributes (start and end time) and their textual content, along with the attribute *pos*, which refers to the part of speech of a word and their id (which should normally have the document on which the word object is stored as a prefix, but we will come back to the subject of explicit and implicit object ids).

For a single observation, objects of the same type are stored in a single file. The syntax file that contained the sentence of our example would look like this:

```
<syntax nite:id="s_0" cat="S">
  <syntax nite:id="s_1" cat="NP">
    <nite:child href="o1.text.xml#w_0"/>
  </syntax>
  <syntax nite:id="s_2" cat="VP">
    <nite:child href="o1.text.xml#w_1"/>
  </syntax>
  <syntax nite:id="s_3" cat="NP">
    <nite:child href="o1.text.xml#w_2"/>
    <nite:child href="o1.text.xml#w_3"/>
  </syntax>
</syntax>
```

From a typical XML perspective, the syntax object with id *s_0* has 3 children, the syntax objects with ids *s_1*, *s_2* and *s_3*, which have the children of which are objects of the type *nite:child*. The *nite:child* objects are also descendants of the initial syntax object.

At this point comes in the difference between typical XML and the data model of NXT. The *nite:child* object of the syntax *s_1* has an *href* attribute, which points to the object of the document *o1.text.xml* that has the id *w_0*. The document *o1.text.xml* has the following form:

```
<word nite:id="w_0" nite:start="0.0"
 nite:end="0.059999998000000006" pos="PPER"> I </word>
<word nite:id="w_1" nite:start="1.100000023"
 nite:end="2.029999971" pos="VVFIN"> cross </word>
<word nite:id="w_2" nite:start="2.029999971"
 nite:end="2.410000085"  pos="ART"> the </word>
<word nite:id="w_3" nite:start="2.410000085"
 nite:end="3.3299999230000004"  pos="NN"> street </word>
```

Therefore, according to the NXT data model, the word object with id *w_0* is also a descendant of the syntax object *s_1* and a descendant of the root syntax object with id *s_0*. Consequently, since timing attributes of an object are inherited by its descendants, we can see that in order to get the start timing of the syntax object *s_0* it is not enough

to access the file on which it is stored. On the contrary, we should access all the files that store descendants of the object, get all their start timings and return their minimum value.

Returning to the diagram above, as far as gestures are concerned, we firstly have the *gtype* tree, that contains general categories of gestures, which themselves contain specific gestures. Each gesture object has the *hands* attribute, its timings attribute and a link relationship of type *type* that connects it with one gtype object. Again we should emphasize on the fact that *gtype* and *gesture* objects are stored on different documents.

2.1.2 Query language

In order to allow its users to perform data search, given that there is no universal data set structure and each user may have their very own needs NXT comes together with a specialist query language, NQL, which is designed to perfectly suit the model used by the platform.

In NQL terminology the basic query unit is a *simple query*, which returns a set of tuples that satisfy a set of conditions. A simple query consists of two parts, separated by colon. In the first part the user should declare the variables that will be used to represent data objects. In the second part the set of conditions, that should be satisfied for an object to be part of the output, is defined. As an example, in the query

($a word) ($b word) : $a@orth == $b@orth && $a != $b && $a@pos == "VVFIN"

on the variable declaration part, on the left of the colon, we have the declaration of two variables, $a and $b which should be both of type *word*. On the condition part of the query, we have the conjunction of three conditions. The query will return tuples that consist of two different words for which their *orth* attribute is equal, and the *pos* attribute of one of them equals to *"VVFIN"*.

Using *simple queries* as a building brick a user may create *complex queries*, which refers to giving the output of one query as an input to another query. This is done by write simple queries side by side separated by two colons (::).

As already mentioned NQL's main target is to cover all the query-related needs that may appear during a research related with natural language processing. To do so it provides a wide variety of operators that are expected to be needed for such tasks. Firstly, there are various query functions, mostly related to object timing information.

Secondly, node property tests allow users to fetch objects that have a specific property. Conditions may contain equality and order tests along with regular expression comparisons. Finally, the language provides operators to test both temporal and structural relations between objects, which are both very important features for the tasks for which the toolkit is expected to be used.

Apart from all the positive aspects, NQL's current implementation has a number of drawbacks. It is old and hence slow, hardly maintained and lacking some very useful operators, such as sisterhood. When it was developed it was also considered memory intensive, however this is not the case today. The scope based on which the language was implemented was to serve efficiently queries that would refer on one observation at a time, and have an acceptable performance when dealing with multiple observations. However, we now believe that because of the advantages of the language it is worth trying to increase evaluation efficiency of the latter class of queries.

2.1.3 A first attempt

Mayo et al. [8] tried to improve the situation, targeting on an implementation that would achieve faster query execution and lower memory requirements. To do so they reimplemented the language on top of XQuery [2], a Turing-complete functional query/programming language, created for querying over sets of XML documents and consists a W3C Recommendation. Their experiments were performed using two different implementations strategies.

Firstly, they used NXT's stand-off format. The main issue related with this approach it the creation of XQuery functions that can navigate through the interleaving tree structure of the data, a process that involves following pointers that point to different documents. The search for ancestors is a computationally expensive operation, since it might require to access every single object in a coding file. Moreover, it might require the evaluation of multiple queries sequentially, depending on the depth of the tree structure. These issues will affect our implementation, as we will see later.

The second strategy of [8], which resulted in promising results in the evaluation process, involved the use of a redundant data representation that derived from the concept of data knitting, which refers to the process of starting from an XML document and recursively follow its child pointers to access all the documents that store objects that are descendants of the initial document's objects, in order to store the full expanded result in a single document. For instance, the knitted document that would

derive from the example of section 2.1.1 would be:

```
<syntax nite:id="s_0" cat="S">
  <syntax nite:id="s_1" cat="NP">
    <word nite:id="w_0" nite:start="0.0"
nite:end="0.059999998000000006" pos="PPER"> I </word>
  </syntax>
  <syntax nite:id="s_2" cat="VP">
    <word nite:id="w_1" nite:start="1.100000023"
nite:end="2.029999971" pos="VVFIN"> cross </word>
  </syntax>
  <syntax nite:id="s_3" cat="NP">
    <word nite:id="w_2" nite:start="2.029999971"
nite:end="2.410000085"  pos="ART"> the </word>
    <word nite:id="w_3" nite:start="2.410000085"
nite:end="3.3299999230000004"  pos="NN"> street </word>
  </syntax>
</syntax>
```

What we did was replace all the *child* objects of the initial *syntax* file with the *word* objects that they point to. Given this document any query related with these objects it contains can be answered by a single XQuery statement.

This representation allows the exploitation of XPath, improving the implementation's efficiency but introduces significantly increased space complexity, because a single element may appear in multiple knitted trees. Moreover, as a consequence of the above this approach introduces the need of pruning duplicate results. Finally, a computationally heavy preprocessing stage is required in order to produced the knitted XML documents.

Our work is closely related with the first strategy. The process of data knitting is equivalent with performing a set of selection and join operations on objects and their descendants. The fact that selections and joins are optimized operators within database systems makes us optimistic that we will achieve good performance without preprocessing and extra space complexity.

In order for our work to be complete we should have also compared our results with the results reported by [8], however this was not possible at the time of delivery of this project, since not all operators used in their tests have been included in our implementation yet. Moreover, it would not be safe to make conclusions from comparing our implementation's runtime against the results of a system that was developed 8 years

ago and tested using the technology that existed then.

2.2 Other similar tools

There is a wide variety of natural language annotation tools, each of them focusing on a different target. There have been many efforts for creating intuitive and user-friendly tools, sacrificing the complexity of the tasks they can perform to become easier to use. *Brat*[11] and the *Glozz Platform* [14] are two such examples.

Brat[11] is a web-based tool for NPL-assisted text annotation. Its main focus is to provide an intuitive and user friendly interface that allows the combination of NLP technology with human intervention. It is a browser-based application that uses the most standard human-computer interaction mechanisms such as dragging and double-clicking, allowing users to easily annotate and visualize their data. However, there is a trade-off between ease of use and complication of tasks that it can perform and therefore it is unable to support a complex annotation model, an area where NXT shines.

The *Glozz Platform* [14] is another corpus annotation tool with a set of salient features. It order to be generic and support heterogeneous objects its annotation technique is based on an abstract meta-model, similar to the data model of NXT. Moreover, it comes together with a highly intuitive and well developed user interface and a powerful query language. On the other hand, the *Glozz platform* is only designed for text annotation, and cannot integrate any other data sources, such as gestures, or blood pressure data.

POWLA [7] constitutes a generic formalism to represent linguistic annotations which is not strictly connected with a specific format of annotation layers, in order to deal with annotations produced by different platforms. Unlike NXT it does not contain its own special-purpose data representation format but employs standards with wide support. However,the *NLP Interchange Format(NIF)* it uses up to now can only support morphosyntactic and syntactic annotations, limiting its functionality when it comes to more complex types of annotations.

FreeLing [9] is another open-source multilingual language processing library. Even though it has rich and efficient functionality in language processing (tokenizing, lemmatizing, part-of-speech tagging etc) and annotation it provides no mechanism for querying.

OPUS [13] is a collection of tools and interfaces capable of collecting a wide range

of parallel documents and after preprocessing them produce a set of parallel corpora in a format that will then be useful for various applications such as statistical machine translation and multilingual terminology extraction. Similarly to NXT, data is represented in XML format. However, the target of the two systems is completely different, since for *OPUS* the word parallel refers to documents in different languages and therefore we do not have different kind of signals that occur in parallel, as in NXT.

As far as searching tools are concerned, it is difficult to fulfil the needs of linguistic engineers with a general-purpose query language and because of that various query languages have been proposed to deal with that. Some of them assume that the data can be expressed as a single ordered tree [1, 10] or provide limited support for links outside the tree [3]. [6] proposed a technique for answering queries related with linguistics. Data is represen in the form of data cubes with time dimension and use OLAP operations are used to perform the queries, producing a short description that summarize the query results. The supported operators, however, cannot support as in-depth data analysis as required. Finally, [12] propose a natural language query interface, with special care to challenges introduced by the the nature of natural language such as language ambiguities and ill-formed queries. Moreover, its focus is on structured data, rather than the semi-structured, XML-based approach of NXT.

From all the above we conclude that even thouth NXT was initially presented a long time ago, its combination of salient features allows it to still be used by linguists.

Chapter 3

XML database selection

As already mentioned the core idea behind our reimplementation of NQL would be to implement it on top of an existing XML database, since such systems offer a set of salient features. Firstly, an XML database includes thte advantages of a relational database such as using schemata, which in XML refer to XSchema or DTDs , and query languages such as XPath or XQuery. Moreover, XML databases minimize the need of extracting information from metadata files, since the entire content store becomes metadata by the use of XPath or Xquery. Finally, common XML related queries, such as sisterhood have been optimized and consequently can be performed very efficiently.

There is a number of different XML database systems and therefore in order to begin our implementation we should decide which system we will use. After researching for the solution that best fits our needs we ended up in a dilemma between eXist-db[1] and MarkLogic[2].

3.1 eXist-db

eXist-db is an open source native XML database system which supports the most significant standards and technologies such as XPath, XQuery, REST and SOAP. It is capable of saving data related with common applications such as XForms very easily. Among its benefits is that its installation is very simple and it offers a very user friendly interface, making it ideal for new users with a little experience. As far as performance is concerned, it offers a variety of index types which can be used according to each use case's needs to boost performance. Moreover, it automatically builds indexes using

[1]http://exist-db.org/

[2]http://www.marklogic.com/

a keyword indexing system, making it trivial to perform high-performance keyword search on documents.

3.2 MarkLogic

On the other hand, MarkLogic is a commercial XML (and recently NoSQL) database. As a commercial product MarkLogic server offers a more complete set of features such as data replication, rollback, point-in-time recovery and direct connection on Hadoop Distributed File System (HDFS). Moreover, its developer licence, which is free, covers our needs more than adequately since it contains 1TB of data storing along with universal indexing, REST and Java APIs and disaster-recovery features.

3.3 Evaluation

Since both systems cover our needs, in order to decide between the systems we decided to perform a quantitative comparative evaluation of both systems. We wanted to test which system can more efficiently perform insertions and deletions of large data sets in batch and querying over them, with and without the use of indexes.

3.3.1 Insertions/deletions

The first test we performed was to insert large amounts of data in batch to both systems. Batch insertions will be important for our language's implementation since whenever a future user comes across a new data set they will first need to insert it in the database to start performing queries over it. In order to measure scalability we decided to start from small 100MB data sets and gradually increase it up to 2GB of data. The data set we used was downloaded from *TPoX XML Data*[1].

Table 1 summarizes the performance of both systems as far as insertions and deletions are concerned. MarkLogic outperforms eXist-db when it comes to small data set insertions, but for larger data sets the time differences become relatively smaller and for the largest 2GB data set eXist-db performed a lot better. However this is not the case in deletions. Here, MarkLogic's performance is way better. This happens because eXist-db when instructed to delete a document it actually starts deleting it from the disk and the system can be again accessed when it is done. However, MarkLogic just

[1]http://tpox.sourceforge.net/tpoxdata.htm

	100MB	200MB	500MB	1GB	2GB
eXist-db	83.4s	181.6s	480.5s	925.2s	2071.2s
	40.8s	81.6s	218.7s	583.8s	1626.9s
MarkLogic	33.5s	132.8s	454.3s	843.4s	2528.2s
	1.5	2.2s	2.4s	2.4s	2.4s

Table 1: Insertion (first row) and deletion (second row) time in seconds

marks the document as deleted without immediately removing it from disk, allowing the system to perform deletes very efficiently.

3.3.2 Query evaluation

Even though insertions and deletions will be a factor that will be taken into consideration on the system we will choose, their importance is way inferior compared to queries, because insertions/deletions will not occur frequently when using NXT. For instance, a normal use case would contain only one insertion of a data set along with multiple queries over it. Therefore, the two systems' performance on the different queries we will execute will most definitely define which system we are going to use.

We present the results of our experiments on table 2. In order for the reader to better understand the tasks performed by each query we provide a description of the query rather than its exact code.

3.3.3 Result evaluation and conclusions

It is obvious that for most queries MarkLogic out performed eXist-db significantly. However, there is a set of observations that have to be made in order to draw the correct conclusions from our experiments.

Firstly, table 2 only contains the performance of MarkLogic with the use if indexing. This is because MarkLogic automatically creates indexes whenever the user inserts any data set in it. Therefore, even though the automatically constructed indexes may not be optimal, lead to a significant boost of performance in comparison to the case in which no indexes are used.

As far as eXist-db is concerned, the automatically constructed indexes the system creates did not help with the queries we performed. Moreover, even though the system provides a wide range of index types, configuring them is not trivial. Furthermore,

Query type	eXist-db (no index)	eXist-db (with index)	MarkLogic (with index)
selection over 1mil element table	11.834s	0.005s	0.506s
selection, projection and output formatting over 100k element table	35.592s	0.005s	1.223s
selection of one element from 100k element table and join with 250k element table	2.63s	NA	0.622s
Element sisterhood on 1 mil element table	55.559s	NA	6.363s
Single element insertion on 1 mil element table	173.61	NA	75.096s
Single element deletion on1 mil element table	18.68	NA	94.092s

Table 2: Query evaluation time in seconds

even if an index is successfully constructed, it might not be used by the system even if the query is related to the attribute on which the index was constructed. To make it more clear we will use an example.

In order to select an *order* element the *id* of which is 934330 we could either just use XPath (*//Order[@ID ="150985"]*) or use a FLWOR (for, let, where, order by, return) XQuery statement (*for $ord in//Order where $ord/@ID = "934330" return $ord*). The eXist-db engine attempts to rewrite any query to simple XPath in order to optimize it. However, because FLWOR statements can be quite complex quite frequently the optimization fails and therefore for the FLWOR statement we mentioned the index of the id attribute of orders is not used, making the system's performance significantly worse. In a system were the queries would be performed manually this would not be an important problem but since the system we will build will automatically map NQL to XPath or XQuery statements there is no guarantee that the produced queries would be able to be optimized by eXist-db's engine, especially considering that FLWOR-type queries will be used heavily. This is the reason why most values of the eXist-db (with index) column of table 2 are not available.

However, it would be unfair not to mention that for the queries that eXist-db's engine managed to use an index its performance was superior to MarkLogic's. This is a very impressive result, considering that we are comparing an open source project with a commercial product.

To sum up, we have decided to use MarkLogic for our project, since eXist-db's current version does not support adequately the functionality we require. In the future we might reconsider, in case a future version of eXist-db provides better index configuration and more sophisticated query optimization.

Chapter 4

Reimplementing NQL on top of MarkLogic

After deciding that MarkLogic will be the system on top of which we will reimplement NQL, our next step was to find out what infrastructure it provides in order to do so.

4.1 MarkLogic Server and Java API

As stated in MarkLogic Server's developer technical overview[1] there are three main terms to be clarified in order to administer the server. On the lower level we have the *documents* that contain the actual data. Documents are stored in *forests*, named like that due to the fact that XML documents have a tree structure. One or more forests constitute a database, which is a logical unit against which a user can set up a server of various kinds, such as HTTP and XDBC.

To have data exchange between the client, which in our case will be the Java application layer of NXT, and the MarkLogic server there are two requirements. Firstly, the client should use the XCC interface, which has a set of client libraries that are used to develop applications that communicate with MarkLogic Server. Secondly, an XDBC server should be configured on the MarkLogic Server side, so that XCC connects to its specified port in order to submit its requests. After the XDBC server receives a request, such as an XQuery statement, it returns its results to the XCC-enabled client that submitted the request. Finally the client, again using XCC libraries can further process the results.

[1] http://developer.marklogic.com/learn/technical-overview

Therefore, our first task before beginning the actual coding was to set up an XDBC server in MarkLogic server and connect the NXT search application to it using XCC.

4.2 General idea

Now that we have set up our server and connected to it we are ready to begin our implementation of NQL. In order to submit NQL queries in MarkLogic Server we need to extract their semantics and map them to a form that is supported by the server. As already mentioned, querying in MarkLogic server is performed using XQuery.

In order to discover potential strategies that would allow us to map NQL to XQuery statements we decided to evaluate a set of NQL queries and manually produce their XQuery equivalents. Table 3 shows the equivalences.

NQL	XQuery
($a)	for $a in //* return $a
($a word) ($b word) : $a@pos == $b@pos && $a != $b	for $a in //word, $b in //word where $a/@pos = $b/@pos and not ($a is $b) return ($a, $b)
($a syntax) ($b word) : $a ^$b	for $b in //word, $a in $a/ancestor-or-self::* where fn:compare($a/name(), ”syntax”) = 0 return ($b,$a) (then perform joins to get parents that referenced the element)
($a) : (TIMED($a))	for $a in //* where exists($a/@nite:start) and exists($a/@nite:end) return $a (then perform joins on the href attribute of ancestors and id attribute of descendants to get all the reachable nite:start and nite:end values)

Table 3: NQL to XQuery mappings

To make the above table clear we should explain what each NQL query does. The

first one is a trivial query that binds objects of all the object types of the corpus to variable $a and returns all of them matching objects, without any condition.

The second query returns combinations of two word objects, bound to $a and $b respectively that make the conditions that follow true. The @ symbol gives access to a specific attribute of an object, for instance $a@pos returns the pos attribute of objects matched by $a.

The third query bounds to $a all syntax type objects and to $b all word type objects and then uses the domination operator ^, which returns true if the object on the right of it is a descendant of the object on the left.

Finally, the last query bounds to $a all the object types of the corpus and uses the function TIMED, which returns true for elements that match $a and have start and end attributes. We will present in more detail the operators we have reimplemented on the next section.

The first observation we can make from the above table is that the structure of NQL queries matches well with XQuery FLWOR statements. More precisely, the variable declaration part of the NQL query matches with the *for* statement of XQuery and the condition part of NQL matches with the *where* statement of XQuery. For instance,

($a word)($b word)

carries the same semantics as:

for $a in //word, $b in //word

whereas

$a@pos == $b@pos && $a!=$b

is equivalent to

where $a/@pos = $b/@pos and not ($a is $b)

Therefore, the above mentioned mappings define the general strategy we are going to follow.

The second observation we make from the above mappings is that we should take

special care for queries that involve navigating through different documents of our data set, such as timing-related queries. As already mentioned, there already exist two alternative approaches to deal with such queries. One involves data knitting, which means that we need a preprocessing/knitting phase that would store all the information related with any object at a single file. The second approach is building an in-memory re-entrant forest to evaluate this class of queries.

We are introducing a third approach, which involves performing join operations between the *href* attributes of ancestors and the *ids* of descendants. For instance, returning to the simple data set we used in chapter 2 where we had *syntax* types that pointed to *word* types we would need to evaluate *syntax.href* $\bowtie$ *word.id*.

4.3 Operators

For this project we intend to implement 4 classes of operators. (1)Typical query operators provide the basic functionality for any general purpose query language. (2)Functions provide access to some of the key attributes of objects such as *text*, *start* and *end*. (3)Temporal and (4)structural relations allow researchers to answer questions about the two types of ordering that NXT supports.

4.3.1 Typical queries

In this category belong all the queries that (1)make no use of functions and (2)do not require traversing links between objects. For instance, selections, projections and joins belong in this category. The list of supported operators includes negation, conjunction, disjunction and implication.

Given the general strategy we described in the previous section it is trivial to implement such queries. All that needs to be done is map the variable declaration of the NQL statement to the for XQuery statement and the condition part of NQL to the where clause of XQuery. By fetching the whole XML object we can then choose which attributes we want to include in the returning result set.

4.3.2 Functions

The functions supported by NQL's existing implementation are:

- *TEXT($w)*, which returns the text contained by elements matched by $w

- *ID($w)*, which returns the identifier of elements matched by $w
- *TIMED($w)*, which returns true if an element matched by $w has start and end time and false otherwise
- *START($w)*, which returns the start time of elements matched by $w
- *END($w)*, which returns the end time of elements matched by $w
- *DURATION($w)*, which returns the duration of elements matched by $w
- *CENTER($w)*, which returns the temporal center of elements matched by $w

As already mentioned for the evaluation of functions, the information that is stored only in the document of the referenced objects is not enough. The approach we are introducing involves the following steps:

1. Fetch all referenced objects, evaluating alternative conditions, if there are any
2. Check the existence of the attributes we are looking for
3. If they exist, we have enough information to answer the query
4. If not, we look for references to other objects, that would have the form of a typical XML child (of the type *nite:child*) with an *href* attribute
5. Perform a join between the above mentioned *href* values and the *id* values of potential children(are all the objects that are stored in the documents that are described by the prefix of the *href* values). Fetch the result of the join.
6. Return to step 2.

For instance, to evaluate the query,

($a syntax):(TIMED($a)) && $a@id="o1.syntax.xml#s_1"

we would firstly fetch the syntax object with id equal to *o1.syntax.xml#s_1*, looking for *start* and *end* attributes. Given that it does not have such attributes, we would perform a join operation between the *href* attributes of its *nite:child* children and the *id* attributes of its potential children. In case *word* objects pointed to a lower level object (for instance, syllables), we would continue the same process recursively until we either got the results we were looking for or a join returned an empty result sequence.

As for the implementation of joins, we tested two different alternatives. We first attempted to perform them using the simple XQuery join syntax(see below for the preprocessing we do at load time to allow this simple equality test to work), for instance:

for $a in //syntax
for $b in //word
where $a/@href = $b/@id
return ($a,$b)

Secondly, since we anyway access the attributes of the root element (in this case syntax) looking if it has its own *start* and *end* attributes, we stored all its *href* values, so that we can then implement the join as a selection. For instance, in our previous example, for the syntax with id *S_3* we would have:

for $a in //word
where $a/@id = "o1.text.xml#w_2" or $a/@id = "o1.text.xml#w_3"
return $a

The second implementation turned out to be faster in the average case, since we substitute the join with a selection on an attribute for which an index exists.

An issue that came up when implementing functions was related with the incompatibility between *id* and *href* values of the data sets. According to the standard naming policy of NXT the value of a reference attribute should be the concatenation of the name of the document where the referenced object is stored, the hash character and the referenced object's id, which led to comparing the explicit *href* value with the implicit *id*. For instance, on the above used example we had to compare *o1.text.xml#w_1* with *w_1*. Even though XQuery provides a function to access the part of the *href* value after the # character, using such a function prevented MarkLogic Server from using the index it had built on this specific attribute, reducing significantly the performance of the system. To resolve the issue we decided to have a simple preprocessing step. Before uploading the data to the server we accessed all *id* attributes and converted them from implicit (*w_1*) to explicit (*o1.text.xml#w_1*). Therefore all queries used the explicit representation of the *id* values.

4.3.3 Temporal relations

The temporal relations supported by NQL's existing implementation are:

- *left and right overlap*
- *left and right aligned with*
- *inclusion*
- *same extent as*
- *overlap*
- *contact*
- *precedence*

The names of the relations are self-explanatory, but NQL's documentation provides full description for each one.

To implement temporal relations we followed the same idea as we did with functions. All we have to do here is get the *start* and *end* time of all referenced objects as mentioned above and return object pairs as defined by each temporal relation in NQL's documentation[1].

However, the runtime of the above mentioned implementation was severely increased, as we saw during the experiments we performed. That was expected, since practically we execute the workload of any function query twice, once for every element type that is referenced in the temporal relation condition. Moreover, there is a significant runtime overhead introduced by the fact that we have to submit more queries to the server. In order to improve the system's performance we merged the queries that are executed for the two participants of the query. For instance, in order to get pairs of time-overlapping *syntax* (that inherit their timings from word) and *game* (that inherit their timings from move) elements, which in NQL would be expressed as:

($a syntax) ($b game) : $a overlaps.with $b

we firstly fetched all *syntax* and *game* elements in the same query and then processed the results of it appropriately, in order to separate the elements of each type. We will use an example to describe the query merging process. Firstly, in order to evaluate the above query we would execute the following XQueries sequentially:

for $a in //syntax
for $b in //word

[1]http://groups.inf.ed.ac.uk/nxt/documentation/ar01s05.html

where $a/@href = $b/@id
return ($a,$b)
for $a in //game
for $b in //move
where $a/@href = $b/@id
return ($a,$b)

After merging these queries we would only need to execute:

for $a in //syntax | //game
for $b in //word | //move
where $a/@href = $b/@id
return ($a,$b)

separate the results and then apply the appropriate temporal relation. The runtime of the merged query was reduced compared to the total performance of the two separate queries. Moreover, the preprocessing we did in our data guarantees that two different objects cannot have the same id, so we cannot have collisions such as matching syntax with move objects in the previous example. This small optimization boosted the performance of temporal relation queries significantly.

4.3.4 Structural relations

According to NQL's documentation there are three supported structural relation operators in current NQL's implementation:

- *Identity*(=), returns true if two elements are identical and false otherwise.
- *Dominance*(^), element A dominates over element B if B is a descendant of A.
- *Precedence*(<>), returns true if two elements have the same ancestor and false otherwise.

The implementation of identity relations was trivial, since the require no traversing different XML documents.

To implement the dominance relation we had two options, either begin from the dominating element and move forward towards its children or start from the dominated element and move up the tree towards its ancestors. It is clear, however, that starting from the dominating element is simpler, because, as already mentioned, the

values of *href* attributes contain the document on which the referenced child is stored, whereas when moving upwards from a child to a parent we have no information about which documents we should search for potential parents, and therefore the search space becomes large, damaging the performance of the system. Moreover, we have already implemented the basic functionality for navigating from a parent to a child node for functions and temporal relations.

Despite that, when it comes to precedence relations we have no choice but to move upwards from both participating element types in order to find a common ancestor. If we just searched for parents throughout the whole data set the performance of the system would be extremely slow. Therefore, we need an alternative to limit the search space.

As stated in [8] information about where to look for ancestors is stored in the meta-data file of a data set. For instance, the meta-data XML file element:

```
<coding-file name="syntax">
<structural-layer name="syntax-layer"
 recursive-draws-children-from="words-layer" />
</coding-file>
```

informs us that elements of the type *syntax* draws children from the *words-layer*, which according to:

```
<coding-file name="text">
<time-aligned-layer name="words-layer">
<code name="word"/>
</time-aligned-layer>
</coding-file>
```

refers to *word* type elements. Therefore, when loading the data set we create a simple meta-data structure that maps each object to objects that may be its ancestors. Our approach is not optimal, since there might be the case that an object has multiple possible ancestors, some of which might be irrelevant with a specific query, but will be included in the search space during the evaluation of it. Moreover, large corpora, such as the ones we used for our experiments have multiple documents for each object type, that may refer to different observations. Using all those documents increases the

search space significantly and needlessly. For instance, in the syntax/word example when our query is about words from one specific observation it is wasteful to include in our search space the syntax documents from all the other observations.

Even so, limiting the search space for parent discovery improves vastly our system's performance for structural relation queries, and more specifically for precedence relations.

Chapter 5

Tests

Our comparative evaluation of the former and our own implementation of NQL will consist of two parts.

We will firstly compare the behaviour of the two implementations when they are dealing with a single observation. We remind that an observation is the description of a complete interaction, such as a dialogue or a group discussion. Evaluating queries efficiently when dealing with a single observation was one of the prime objectives of the initial NQL implementation, since it is quite frequent for a researcher to analyse each observation separately.

After that we will test the scalability of the two implementations, by comparing their behaviour when they have to deal with larger data sets. Scalability testing is crucial, since the capabilities of a system are severely limited if it only has acceptable performance with small input data sets.

For both parts we will use a set of 4 simple queries, one from each category. Moreover, in order to make sure that our results are not data set-dependent we run the queries on two different data sets, the HCRC Map Task Corpus[1] which consists of 128 dialogues (a total of 118.4MB of data) and the AMI Meeting Corpus[2], which consists of 100 hours of meeting recordings (205.6 MB of data).

5.1 Runtime comparison for a single observation

For our single observation tests we used the Search Tool of NXT, a console-like application that comes together with NXT and is very user-friendly. When a user runs the

[1]http://groups.inf.ed.ac.uk/maptask/

[2]http://groups.inf.ed.ac.uk/ami/

search tool they first have to select the data set they want to load and after that they can submit their queries. Moreover, the tool stopwatch that we will use to get the runtime of our queries. The only limitation of the search tool is that only a single observation can be loaded and queried at any time, which at this point is irrelevant to our tests since we are measuring the system's performance for a single observation.

In our version of NXT search, the data we want to use should be loaded to the MarkLogic server before running the tool. On start-up the application connects with the MarkLogic server and when a query is submitted it is parsed by the parser of the tool, translated to the appropriate XQuery statements and submitted to the server for evaluation. Finally, the server returns the results to the application. As already mentioned it is possible that a single NQL query may be converted to multiple XQuery statements. At this point the comparison we are going to perform is between the standard NXT search tool and our version of it that uses our implementation of NQL. In order for the results we will report to be meaningful we run our queries over 10 different observations (the size of each observation varies from 1 to 2.5 MB) and report the average execution time. Moreover, to ease the reader's understanding, instead of reporting the actual query we used we report the queries' type. The actual queries can be found in the appendix section.

The following tables contain the results of our experiments.

Query type	Existing implementation	Our implementation
Selection/projection/join query	0.179s	0.123s
Function containing query	1.43s	3.59s
Temporal relation query	3.63s	7.12s
Structural relation query	0.97s	6.01s

Table 4: Average time of 10 observations - HCRC Map Task Corpus

Query type	Existing implementation	Our implementation
Selection/projection/join query	0.483s	0.137s
Function containing query	1.421s	4.21s
Temporal relation query	1.603s	7.97s
Structural relation query	1.01s	5.01s

Table 5: Average time of 10 observations - AMI Meeting Corpus

Firstly, from the two tables we can derive that our implementation outperforms the existing implementation when it comes to selection/projection/join queries. This was expected, since database systems are tailored to evaluate such queries efficiently.

On the other hand, queries that required the evaluation of multiple selection and join operators were slower in our implementation and in the case of temporal relation queries, where multiple such queries had to be evaluated the execution time of our implementation was significantly worse than that of the existing implementation. However, the XQuery code that was produced by our system and then run against MarkLogic Server was not optimized, and we believe that it is doable to produce ”smarter” translations between NQL and XQuery, in order to significantly boost the overall performance.

It is worth mentioning that the performance of our system is similar on both data sets, and therefore we believe that it is safe to conclude that our implementation is data set-independent.

5.2 Scalability evaluation

Apart from comparing the two implementations for a single observation, we wanted to evaluate how well the two implementations deal with larger data sets. As already mentioned the search tool of NXT can only be used for a single observation at a time. Because of that, we used FunctionQuery, a utility that can be used for outputting the results of a query in tab-delimited format. FunctionQuery can also support a single observation at a time, but since it is a command line tool we can write a script that cycles through all observations one-by-one.

As for our implementation, it is not affected by anything that is related with loading multiple observations. Once again we used our version of NXT search tool, preloading the desired volume of data into MarkLogic server and then querying over it.

We present the results of the scalability test of our implementation.

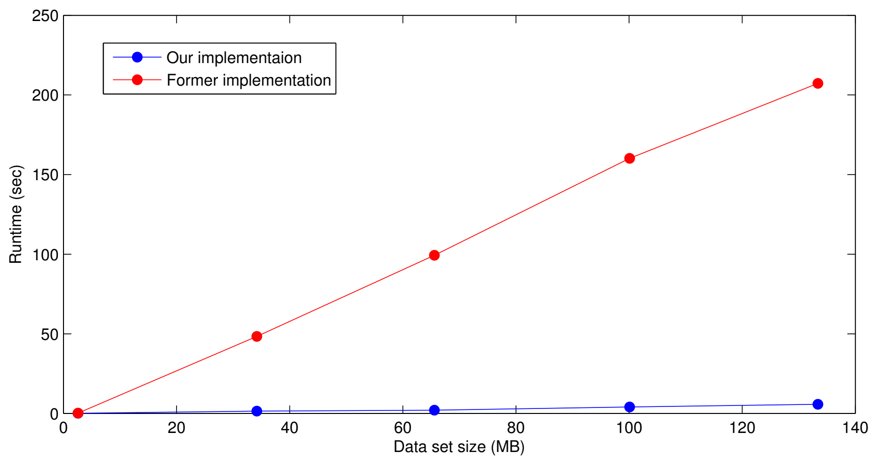


Figure 1: Data set size vs runtime, selection/projection/join query, HCRC Map Task Corpus

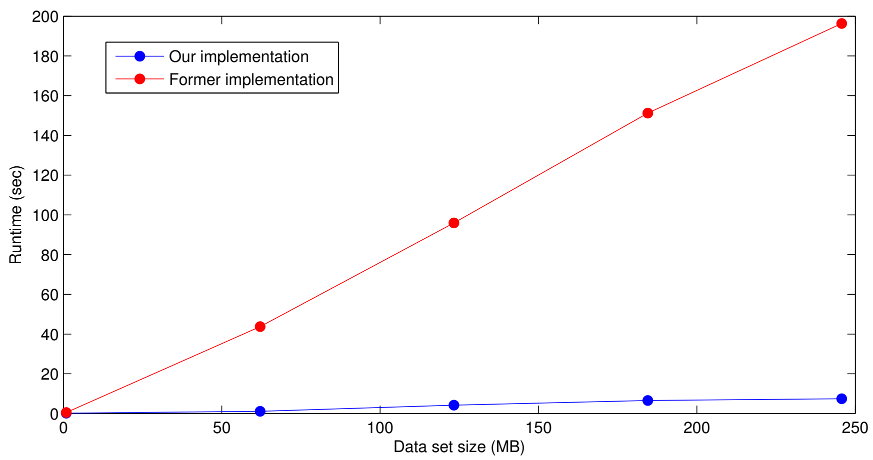


Figure 2: Data set size vs runtime, selection/projection/join query, AMI Meeting Corpus

Firstly, simple selection/projection/join queries are evaluated very efficiently, no matter how large the data set is. For such simple queries that can be evaluated by a single XQuery statement the performance of a database system is incomparable due to the optimization techniques and the use of indexes.

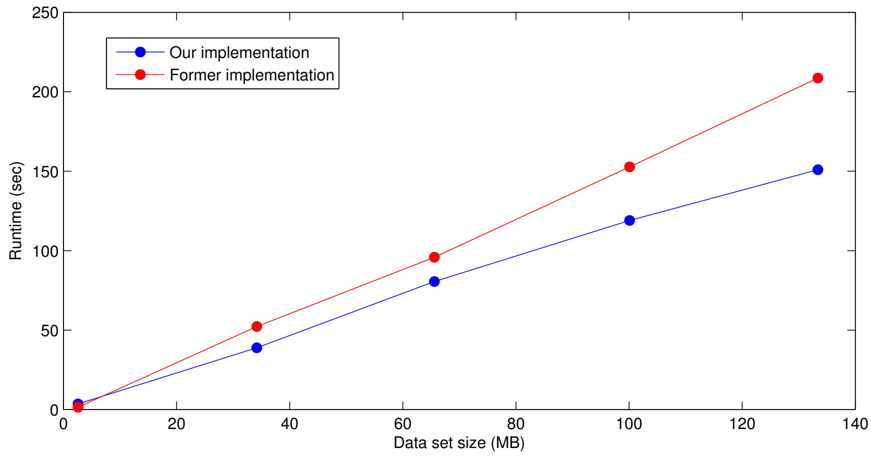


Figure 3: Data set size vs runtime, function containing query , HCRC Map Task Corpus

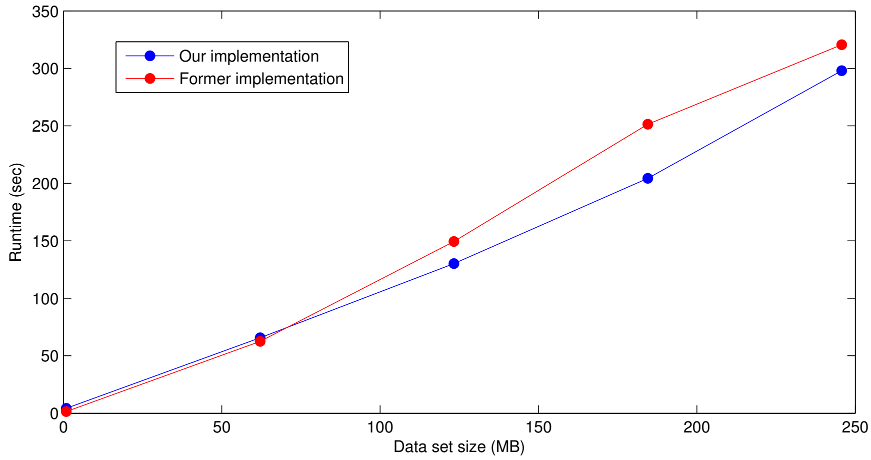


Figure 4: Data set size vs runtime, function containing query , AMI meeting Corpus

Secondly, as far as function containing queries, the results of our implementation are very promising. For small data sets (such as the single observation case we reviewed during in previous section) the former implementation performs significantly better. However, for larger data sets our performance overcomes that of the former implementation. We believe that there are many optimizations that can be applied on top of our implementation to boost its performance even more.

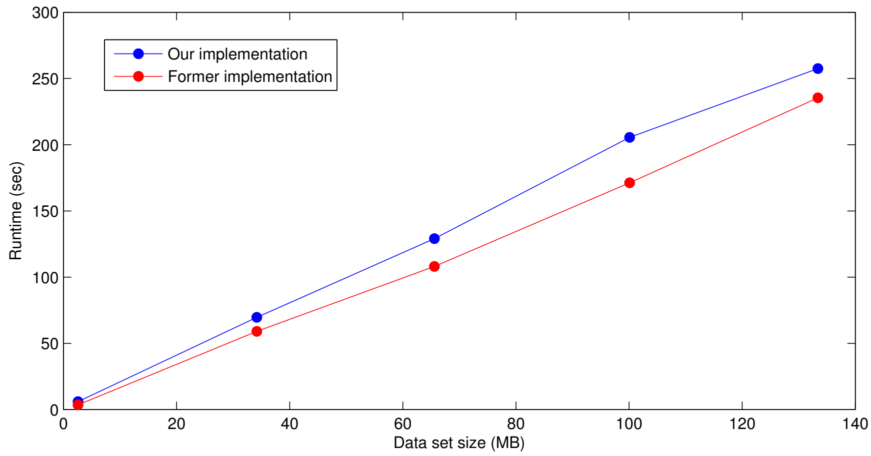


Figure 5: Data set size vs runtime, temporal relation query, HCRC Map Task Corpus

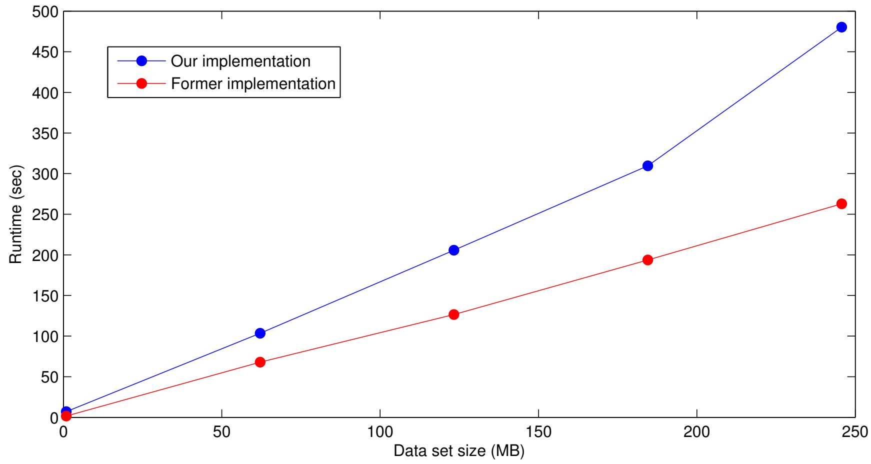


Figure 6: Data set size vs runtime, temporal relation query, AMI meeting Corpus

Thirdly, time relation queries is the only class of queries that the old implementation significantly outperforms us. This result was expected. In our initial implementation of such operators required double the work of any function containing query. After the query merging technique we implemented the situation improved, but there is still a long road until we can say that temporal relation queries can be evaluated efficiently.

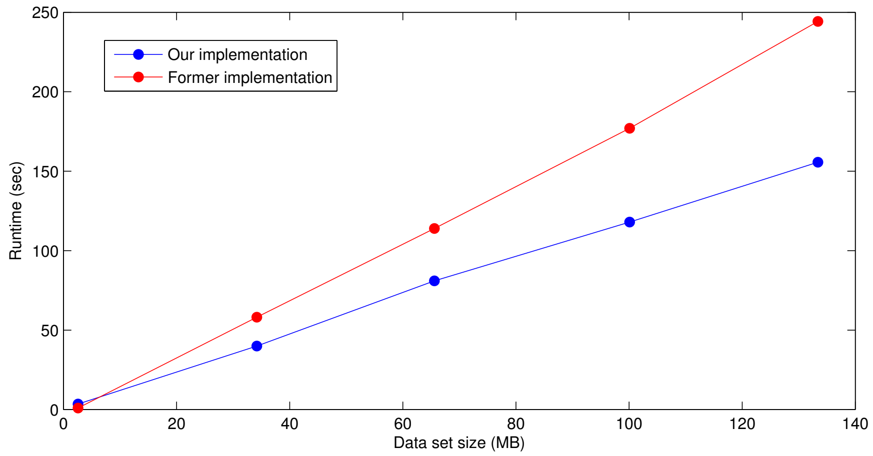


Figure 7: Data set size vs runtime, structural relation query, HCRC Map Task Corpus

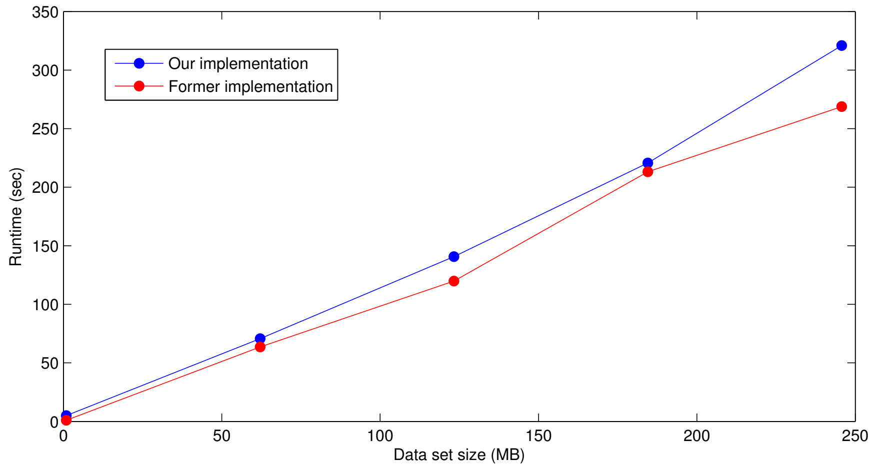


Figure 8: Data set size vs runtime, structural relation query, AMI Meeting Corpus

Finally, the results that derived from the structural relation query were unexpected, since our implementation performed significantly better for the first data set but the former implementation was the winner as far as the second data set is concerned. The reason behind this is that the query we executed in the second data set had a deeper level of recursion, and therefore more XQueries needed to be evaluated to completely answer the query. For such queries the performance of our implementation suffers.

Chapter 6

Conclusion

The NITE XML Toolkit is a set of libraries and tools that allows researchers to ask questions related to natural language corpora, able to support multimodal and heavily cross-annotated data sets. Its annotations have a graph structure that consists of a set if intersecting trees with some superimposed trees. Moreover, it comes with its own query language, NQL, which is tailored to suit the needs of its complex data model. However, NQL's implementation is old and has a number of drawbacks related with its performance, its memory requirements and its maintainability.

In this project we began the process of reimplementing NQL, on top of an existing XML database. We firstly performed a quantitative comparative evaluation of the two most important XML database systems and concluded that MarkLogic Server suits our needs the most. After that we began the reimplementation of NQL, starting from simple selection/projection/join queries and then moving to more complex operators such as functions, temporal and structural relations, which require a navigation though different XML documents in order to access all information needed to answer the query.

6.1 Future work

We believe that our work could become the basis for a wide variety of improvements and extensions. Firstly, apart from further testing/debugging, we believe that it is possible to improve the translation of NQL to XQuery statements in order for the XQueries that are evaluated by the database to be optimized. The queries that derive from our translation are evaluated efficiently, but the fact that we have to run a lot of XQueries for any NQL query that is submitted acts as a bottleneck. We believe that it is possible

to merge some of the automatically produced queries, in order to reduce the overall runtime. For instance, when we first implemented the temporal relation queries, we firstly fetched the timing attributes for the one participant of the query and then for the second one. However, when we merged the process, fetching the timings for both participants at once and then separating them we had a significant boost in performance. Towards that direction we believe that the use of meta-data files could be exploited. In our implementation we used them only to evaluate the precedence structural relation operator. We believe however that using the information contained in those files would help in generating XQuery statements with superior expressive power.

Another possible way to improve our performance was to implement a simple query optimizer, that would ”cleverly” order the evaluation of the operators within our queries. That would result in reducing the search space of the more computationally intense operators.

As for potential extensions, we believe that some operators that would be key for such a system, such as sisterhood, are missing. We intended to implement a sisterhood operator but we faced some issues related with the existing query parser of the system. In the future, we believe that there are plenty of opportunities to add functionality on the existing system.

Finally, we believe that it is worth attempting to implement the data knitting technique on top of an XML database. Such an implementation would introduce some latency related with the pre-process/knitting and upload phase of the execution, along with an increased space complexity. However, after that phase is done, the performance of the system is expected to be impressive, since it will only need to execute simple select/project/join queries, which as we saw from our implementation can be evaluated with great efficiency by an XML database and then prune the duplicate results that will be produced due to the replication of objects that occurs at the knitting process.

Appendix A

Actual queries

We will now present the actual queries we used in our test, along with a short comment on why we selected those queries.

A.1 HCRC Map Task Corpus

For this data set, the simple query we used to evaluate its our systems performance was:

($a move):$a@aban="false_aban" && $a@interj = "false_interj"

A simple selection query, that returns all the *move* objects of the dataset that have their *aban* attribute equal to *"false_aban"* and their *interj* attributes equal to *"false_interj"*. We used this query in order to evaluate our performance when we are querying on a single object type.

The function query we used was:

($a dw):timed($a)

The above query, which returns all the *disfluent word* objects that have both *start* and *end* timing attributes, even though it is simple in conception it is a significant test of our system, because of the fact that there are no conditions limiting the number of *disfluent word* for which we should navigate through their descendant documents.

Under the same circumstances, to cover the temporal relation queries we used:

($a dw) ($b move) : $a overlaps.with $b

The above query returns *disfluent word* and *move* objects the timings of which overlap. Again, the fact that we do not limit the number of objects of both types forces our system to perform the costly inter-document descendant search for every object of the class.

Finally, the structure relation query we used was:

($a dw) ($b tu) : $a ˆ 1$b

that returns pairs of *disfluent word* and *time unit* objects where the *disfluent word* dominates directly to the *time unit*.

A.2 AMI Meeting Corpus

The only significantly different query we used for the second data set was the first query we used:

($a hand)($b movement) :$a@type="off_camera" &&$a@who = $b@who

that joins all the *hand* objects the *type* attribute of which is equal to *off_camera* with the *move* objects on their *who* attribute. We used this query to see how a regular database performs in our implementation.

Other than that, the queries we used here were similar to those we used above. The function query was:

($a dact):(start($a))

which returns the start time of all the *dialogue act* objects, the temporal relation query:

($a dsfl)($b dact):$a overlaps.with $b

which returns pairs of time overlapping *disfluency* and *dialogue act* objects and the structural relation query was:

($a dsfl)($b w):$a ˆ 1 $b

that returns pairs of *disfluency* and *word* objects where the *disfluency* dominates directly to the *word*. We decided to evaluate both data sets with similar queries in order to be able to draw a conclusion about whether our implementation is independent of the data set that is used.

Bibliography

[1] S. Bird, Y. Chen, S. B. Davidson, H. Lee, and Y. Zheng. Designing and evaluating an xpath dialect for linguistic queries. In *Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on*, pages 52–52. IEEE, 2006.

[2] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, J. Siméon, and M. Stefanescu. Xquery 1.0: An xml query language, 2002.

[3] S. Brants, S. Dipper, P. Eisenberg, S. Hansen-Schirra, E. König, W. Lezius, C. Rohrer, G. Smith, and H. Uszkoreit. Tiger: Linguistic interpretation of a german corpus. *Research on Language and Computation*, 2(4):597–620, 2004.

[4] J. Carletta, S. Evert, U. Heid, and J. Kilgour. The nite xml toolkit: data model and query language. *Language resources and evaluation*, 39(4):313–334, 2005.

[5] J. Carletta, S. Evert, U. Heid, J. Kilgour, J. Robertson, and H. Voormann. The nite xml toolkit: flexible annotation for multimodal language data. *Behavior Research Methods, Instruments, & Computers*, 35(3):353–363, 2003.

[6] R. Castillo-Ortega, N. Marín, and D. Sánchez. Linguistic query answering on data cubes with time dimension. *International Journal of Intelligent Systems*, 26(10):1002–1021, 2011.

[7] C. Chiarcos. Powla: Modeling linguistic corpora in owl/dl. In *The Semantic Web: Research and Applications*, pages 225–239. Springer, 2012.

[8] N. Mayo, J. Kilgour, and J. Carletta. Towards an alternative implementation of nxt's query language via xquery. In *Proceedings of the 5th Workshop on NLP and XML: Multi-Dimensional Markup in Natural Language Processing*, pages 27–34. Association for Computational Linguistics, 2006.

[9] L. Padró, M. Collado, S. Reese, M. Lloberes, I. Castellón, et al. Freeling 2.1: Five years of open-source language processing tools. 2010.

[10] D. L. Rohde. Tgrep2 user manual, 2004.

[11] P. Stenetorp, S. Pyysalo, G. Topić, T. Ohta, S. Ananiadou, and J. Tsujii. Brat: a web-based tool for nlp-assisted text annotation. In *Proceedings of the Demonstrations at the 13th Conference of the European Chapter of the Association for Computational Linguistics*, pages 102–107. Association for Computational Linguistics, 2012.

[12] V. Tablan, D. Damljanovic, and K. Bontcheva. *A natural language query interface to structured information*. Springer, 2008.

[13] J. Tiedemann. News from opus-a collection of multilingual parallel corpora with tools and interfaces. In *Recent Advances in Natural Language Processing*, volume 5, pages 237–248, 2009.

[14] A. Widlöcher and Y. Mathet. The glozz platform: a corpus annotation and mining tool. In *Proceedings of the 2012 ACM symposium on Document engineering*, pages 171–180. ACM, 2012.